Producing languages with better abstractions is possible ... why are there not more such languages?
We have acknowledged that present day programming facilities are inadequate ... a good question would be, what do we want?

Today we give two languages that exhibit a high level of parallel abstraction

- ZPL from UW
- Nesl from CMU

The languages have weaknesses, but mostly we’re focused on their strengths as exemplars.
ZPL

- ZPL, a research parallel language w/ 3 goals
  - Performance == as good as platform-specific custom code
  - Portability == runs well on all platforms
  - Convenience == clean, easy-to-understand programs; no parallel grunge
- Developed at UW by 6 really smart grad students: Brad Chamberlain, Sung-Eun Choi, Steve Deitz, E Chris Lewis, Calvin Lin, Derrick Weathersby
The Team

Secosky  Lewis  Snyder  Weathersby
Ngo       Lin    Chamberlain
Deitz     Choi
ZPL is our representative high-level parallel language ... few competitors because achieving those goals is tough.

To realize a solution ... 
- ZPL is designed and built on the CTA
- ZPL is the first high-level language to achieve “performance portability”
- ZPL presents programmers with a visually-cued performance model: WYSIWYG
- ZPL is insensitive to shared or message passing architectures, making it universal

ZPL is “designed from first principles”
Conway’s Game of Life

- Life: organisms w/ 2, 3 neighbors live, birth occurs w/ 3 neighbors; death otherwise; world is a torus
  
  Organism in next generation if position is alive in this generation and has 2 neighbors, or in this generation it has 3 neighbors
  
  Or: (thisGen && neighbors== 2) || (neighbors==3)

See Life As An Array Computation
Count neighbors by adding organisms (bits)

Consider the sum of these arrays from a single cell’s perspective ...
Conway’s Life: The World is bits

[R] repeat

NN := TW@^NW + TW@^N + TW@^NE 
+ TW@^W + TW@^E 
+ TW@^SW + TW@^S + TW@^SE;

TW := (TW & NN = 2) | (NN = 3);

until ! (|<< TW);

Add up neighbor bits

“Or” bits in the world to see if any alive

Apply rules to live by
The torus world says that each direction wraps – expressed as @^n

TW@^nw is the array of Northwest neighbors
```plaintext
program Life;
config const n : integer = 10;
region R = [1..n, 1..n];
direction nw = [-1, -1]; no = [-1, 0]; ne = [-1, 1];
    w = [ 0, -1];
    sw = [ 1, -1]; so = [ 1, 0]; se = [ 1, 1];
var  TW : [R] boolean;
    NN : [R] sbyte;
procedure Life();
begin -- Initialize the world
  [R] repeat
    NN := TW@^nw + TW@^no + TW@^ne + TW@^w + TW@^e
         + TW@^sw + TW@^so + TW@^se;
    TW := (TW & NN = 2) | (NN = 3);  
    until !(|<< TW);
end;
```

Conway's Life

The world is \( n \times n \); default to 10

Index set of computation

Problem state, The World

Work array, Number of Neighbors

Entry point procedure

I/O or other data specification

Region \( R \) ==> apply ops to all indices

Add 8 nearest neighbor bits (type coercion like C); carat(\(^\)) means toroidal neighbor reference

Update world with next generation

Continue till all die out
program Life;
config const n : integer = 10;
region R = [1..n, 1..n];
direction nw = [-1, -1]; no = [-1, 0]; ne = [-1, 1];
    w = [ 0, -1];  e = [ 0, 1];
    sw = [ 1, -1]; so = [ 1, 0]; se = [ 1, 1];
var  TW : [R] boolean;
    NN : [R] sbyte;
procedure Life();
begin -- Initialize the world
[R] repeat
    NN := TW^nw + TW^no + TW^ne
        + TW^w +          TW^e
        + TW^sw + TW^so + TW^se;
    TW := (TW & NN = 2) | (NN = 3);
    until !(|<< TW);
end;
Regions, A Key ZPL Idea

- Regions are index sets ... not arrays
- Any number of dimensions, any bounds
  
  region V = [1..n];
  region R = [1..m, 1..m]; BigR = [0..m+1, 0..m+1];
  region Left = [1..m, 1];
  region Odds = [1..n by 2];

- Short names are preferred--regions are used everywhere--and capitalization is a coding convention
- Naming regions is recommended, but literals are OK
Regions are used to declare arrays ... it’s like adding data to the indices

Capitals are used by convention to distinguish arrays from scalars

Named or literal regions are OK

```plaintext
var A, B, C : [R] double;
var Seq : [V] boolean;
var Huge : [0..2^n, -5..5] float;
```

Regions are used once; no array has more than one region component

Regions are a source of parallelism...
Statements containing arrays need a region to specify which items participate
\[1..n,1..n\] A := B + C;
[\[R\]] A := B + C; -- Same as above

Regions are scoped
[R] begin
  ...
[Left] ...
end; [Left]

All array computations in compound statements are performed over indices in [R], except statement prefixed by [Left]

Operations over region elements performed in parallel
Let $A$, $B$ be arrays over $[1..n,1..n]$, and $C$ be an array over $[2..n-1,2..n-1]$ as in

```plaintext
var A, B : [1..n,1..n] float; C : [2..n-1,2..n-1] float;
```

Then

```plaintext
[2..n-1,2..n-1] A := C;
```

```plaintext
[2..n-1,2..n-1] C := A + B;
```

```plaintext
[2..n-1,2] A := B;
```
The @ operator combines regions with directions to allow references to neighbors
- Two forms, standard(@) and wrapping(@^)
  - Syntax: A@east    A@^east
  - Semantics: the direction is added to elements of region giving new region, whose elements are referenced; think of a region translation
    \[1..n,1..n\] A := A@^east; -- shift array left with wrap around

- @-modified variables can appear on l or r of :=
Let

\[ \text{var } A, B : [1..n,1..n] \text{ float; } C : [2..n-1,2..n-1] \text{ float; } \]
\[ \text{direction east} = [0,1]; \text{ ne} = [-1,1]; \]

Then

\[ [2..n-1,2..n-1] A := C[^{\text{east}}]; \]
\[ [2..n-1,2..n-1] A := C[^{\text{ne}}] + B[^{\text{ne}}]; \]
\[ [2..n-1,2] A^{\text{east}} := B; \]
Reductions, Global Combining Operations

- Reduction (<<) “reduces” the size of an array by combining its elements
- Associative (and commutative) operations are +<<, *<<, &<<, |<<, max<<, min<<

\[
\begin{align*}
[1..n, 1..n] \text{ biggest} & := \text{max}<<A; \\
[R] \text{ all_false} & := |<< TW;
\end{align*}
\]

- All elements participate; order of evaluation is unspecified ... caution floating point users
- ZPL also has partial reductions, scans, partial scans, and user defined reductions and scans
Operations On Regions

- The importance of regions motivates region operators
- Prepositions: at, of, in, with, by ... take region and direction and produce a new region
  - at translates the region’s index set in the direction
  - of defines a new region adjacent to the given region along direction edge and of direction extent

```plaintext
region R = [1..8,1..8];
C = [2..7,2..7];
var X, Y : [R] byte;
direction e = [ 0,1];
n = [-1,0];
ne = [-1,1];
```

```
[R]X:=
[C]X:=
[C at e]Y:=
[n of C]Y:=
[C]Y:=X@ne
```

[Execution Diagram]
Applying Ideas: Jacobi Iteration

- Model heat defusing through a plate
- Represent as array of floating point numbers
- Use a 4-point stencil to model defusing
- Main steps when thinking globally
  
  Initialize
  Compute new averages
  Find the largest error
  Update array
  … until convergence

High-level Language should match high-level thinking
“High Level” Logic Of J-Iteration

program Jacobi;
config var n : integer = 512;
  eps : float = 0.00001;
region     R = [1..n, 1..n];
  BigR = [0..n+1,0..n+1];
direction N = [-1, 0];  S = [ 1, 0];
    E = [ 0, 1];  W = [ 0,-1];
var     Temp : [R] float;
  A : [BigR] float;
err : float;

procedure Jacobi();
[R] begin
[BigR]  A := 0.0;
[S of R] A := 1.0;
repeat
    Temp := (A@N + A@E + A@S + A@W)/4.0;
    err  := max<< abs(Temp - A);
    A    := Temp;
until err < eps;
end;
ZPL has ‘full’ reduce: +<<, *<<, max<<, ...
ZPL also has ‘partial’ reduce
- Applies reduce across rows, down columns,...
- Requires two regions:
  - One region on statement, as usual
  - One region between operator and operand

\[[1..n,1] B := +<< [1..n,1..n] A;\] -- add across rows
\[[1,1..n] C := \text{min}<<[1..n,1..n] A;\] -- min down columns
- In these examples, result stored in 1st row/col

Collapsed dimensions indicate reduce dimension(s)
Flood -- Inverse of Partial Reduce

- Reduce “reduces” 1 or more dimensions
- Opposite is flood -- fill 1 or more dimensions

\[
[1..n, 1..n] \ B := >> [1..n, 1] \ A;
\]
\[
[1..n, 1..n] \ B := >> [1..n, n] \ A;
\]

- The replication uses multicast, often an efficient operation
Closer Look: Scaling Each Row

\[ [1..m,1] \text{ MaxC } := \max<[1..m,1..n] A; \quad \text{Max of each row} \]
\[ [1..m,1..n] \quad A := A / >>[1..m,1] \text{ MaxC}; \quad \text{Scale each row} \]

- Flooding distributes values (efficiently) so that the computation is element-wise ... lowers communication

\[
\begin{array}{cccc}
2 & 4 & 4 & 2 \\
0 & 2 & 3 & 6 \\
3 & 3 & 3 & 3 \\
8 & 2 & 4 & 0 \\
\end{array}
\begin{array}{cccc}
4 & 4 & 4 & 4 \\
6 & 6 & 3 & 6 \\
3 & 3 & 3 & 3 \\
8 & 8 & 8 & 8 \\
\end{array}
\]

Keep MaxC a 2D array to control allocation
Flood dimensions recognize that specifying a particular column **over specifies** the situation
Need a *generic* column -- or a column that does not have a specific position ... use ‘*’ as value

```plaintext
region FlCol = [1..m, *];       -- Flood regions
    FlRow = [* , 1..n];
var    MaxC : [FlCol] double;  -- An m length col
    Row : [FlRow] double;     -- An n length row
[1..m,*] MaxC := max<< [1..m,1..n] A; -- Better
```

Think of column in every position
Since flood arrays have some unspecified dimensions, they can be “promoted” in those dimensions, i.e logically replicated

- Scaling a value by max of row w/o flooding:

\[
[1..m,\ast] \quad \text{MaxC} := \text{max}\ll [1..m,1..n] \ A;
\]

\[
[1..m,1..n] \quad A := A / \text{MaxC}; \quad \text{-- Scale A;}
\]
Flood v. Singleton Difference

- Lower dimensional arrays can specify a singleton or a flood ... it affects allocation

- Region \([1..n,1..n]\) allocated to 4 processors

- Regions \([1..n,1]\) and \([n,1..n]\) allocated to 4 processors

- Regions \([1..n,*]\) and \([*,1..n]\) allocated to 4 processors
For each col-row in the common dimension, flood the item and combine it...

\[
\begin{align*}
\text{var} & \quad A: [1..m, 1..n] \text{ double;} \\
& \quad B: [1..n, 1..p] \text{ double;} \\
& \quad C: [1..m, 1..p] \text{ double;} \\
& \quad \text{Col: [ 1..m,*] double;} \\
& \quad \text{Row: [*, 1..p] double;} \\
\end{align*}
\]

\[
[1..m,1..p] \quad C := 0.0; \quad \text{-- Initialize C}
\]

\[
\text{for k := 1 to n do}
\]

\[
[1..m,*] \quad \text{Col := >>[ ,k] A;} \quad \text{-- Flood kth col of A}
\]

\[
[*,1..p] \quad \text{Row := >>[k, ] B;} \quad \text{-- Flood kth row of B}
\]

\[
[1..m,1..p] \quad C += \text{Col*Row}; \quad \text{-- Combine elements}
\]

\[
\text{end;}
\]
## SUMMA, The First Step

<table>
<thead>
<tr>
<th>Col</th>
<th>Row</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a11</td>
<td>a11</td>
<td>a11</td>
</tr>
<tr>
<td>a21</td>
<td>a21</td>
<td>a21</td>
</tr>
<tr>
<td>a31</td>
<td>a31</td>
<td>a31</td>
</tr>
<tr>
<td>a41</td>
<td>a41</td>
<td>a41</td>
</tr>
<tr>
<td>b11</td>
<td>b12</td>
<td>b13</td>
</tr>
<tr>
<td>b11</td>
<td>b12</td>
<td>b13</td>
</tr>
<tr>
<td>b11</td>
<td>b12</td>
<td>b13</td>
</tr>
<tr>
<td>b11</td>
<td>b12</td>
<td>b13</td>
</tr>
</tbody>
</table>

\[
\begin{array}{ccc}
    c11 & c12 & c13 \\
    c21 & c22 & c23 \\
    c31 & c32 & c33 \\
    c41 & c42 & c43 \\
\end{array}
\times
\begin{array}{ccc}
    a11 & a12 & a13 & a14 \\
    a21 & a22 & a23 & a24 \\
    a31 & a32 & a33 & a34 \\
    a41 & a42 & a43 & a44 \\
\end{array}
=
\begin{array}{ccc}
    b11 & b12 & b13 \\
    b21 & b22 & b23 \\
    b31 & b32 & b33 \\
    b41 & b42 & b43 \\
\end{array}
\]

SUMMA is the easiest MM algorithm to program in ZPL

ván de Geijn & Watts say it’s the fastest machine independent
For each col-row in the common dimension, flood the item and combine it...

\[
[1..m,1..p] \quad C := 0.0; \quad -- \text{Initialize } C \\
\text{for } k := 1 \text{ to } n \text{ do} \\
[1..m,*] \quad \text{Col := } >>[ ,k] \text{ A; } \quad -- \text{Flood kth col of A} \\
[*,1..p] \quad \text{Row := } >>[k, ] \text{ B; } \quad -- \text{Flood kth row of B} \\
[1..m,1..p] \quad C += \text{Col*Row; } \quad -- \text{Combine elements} \\
\text{end;}
\]

--- or, more simply ---

\[
\text{for } k := 1 \text{ to } n \text{ do} \\
[1..m,1..p] \quad C += (>>[ ,k] \text{ A})*(>>[k, ] \text{ B}); \\
\text{end;}
\]
If flooding is so good for columns/rows, why not use it for whole planes?

region IK = [1..n,* ,1..n];
   JK = [* ,1..n,1..n];
   IJ = [1..n,1..n,* ];
   IJK = [1..n,1..n,1..n];

[IJ] A2 := >>A#[Index1, Index2];
[JK] B2 := >>B#[Index2, Index3];
[IK] C := +<<[IJK](A2*B2);
Partial scans are possible too, but unlike reduce they do not reduce dimensionality, so the compiler cannot tell which dimension to reduce ... so specify

\[
+ | | [2] A \text{ is a partial scan in the 2nd dimension}
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\Rightarrow
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

\[
+ | | [2]
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\]
The operators for reduce, scan and flood are suggestive ...

- Reduce $\ll$ produces a result of smaller size

- Scan $|$ $|$ produces a result of the same size

- Flood $>>$ produces a result of greater size
ZPL comes with “constant arrays” of any size

Index\(i\) means indices of the \(i^{\text{th}}\) dimension

\[
[1..n,1..n]\begin{align*}
Z & := \text{Index1}; \quad \text{-- fill with first index} \\
P & := \text{Index2}; \quad \text{-- fill with second index} \\
L & := Z=P; \quad \text{-- define identity array}
\end{align*}
\text{end;}
\]

Index\(i\) arrays: compiler created using no space

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 2 & 3 & 4 \\
2 & 2 & 2 & 2 & 1 & 2 & 3 & 4 & 0 \\
3 & 3 & 3 & 3 & 1 & 2 & 3 & 4 & 0 \\
4 & 4 & 4 & 4 & 1 & 2 & 3 & 4 & 0 \\
\end{array}
\]

\begin{array}{cccc}
\text{Index1} & \text{Index2} & L
\end{array}
\]
The remap operator (#) implements general data motion, including rank change

- Two cases:
  - Gather, $A := B[C_1, C_2]$;
  - Scatter, $A[C_1, C_2] := B$;

- For $r$ rank array, provide $r$ rank $r$ arrays giving indices to be referenced.

- Transpose: $AT[i, j] = A[j, i]$

  $[R] AT := A[Index_2, Index_1]$; -- Standard Idiom for transpose
Remap (Gather)

The index array in the ith position gives the indices for the ith dimension

\[ a c e b d f \Leftrightarrow a b c d e f \# [1 3 5 2 4 6] \]

\[
\begin{array}{cccc}
\text{AT} & \text{A} & \text{Index2} & \text{Index1} \\
\end{array}
\]

\[
\begin{bmatrix}
1 & 2 & 3 & 4 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 2 & 2 & 2 & 2 \\
1 & 2 & 3 & 4 & 3 & 3 & 3 & 3 \\
1 & 2 & 3 & 4 & 4 & 4 & 4 & 4 \\
\end{bmatrix}
\]
Scatter Remap has potential problem in that values can map to the same place ... order is unspecified ... use +=, etc. if not unique

Scatter: For a value, where does it go?
CTA and ZPL

- ZPL was built on the CTA
  - Semantics of operation customized to CTA
  - Compiler targets CTA machines
  - Performance model reflects the costs of CTA
- The benefit of building on the CTA:
  - Programming constraints are realistic, scalable
  - Programs are portable *with performance*
  - Programmers can reliably estimate performance and observe it (or better) on every platform

Building on CTA is a key contribution of ZPL
We now explain ZPL’s performance model

What is it?

- It is the way programmers know how fast (or slow) the statements of their programs will run.”

- We all “know” the performance model for C

- Every || language should have a performance model

- Learning this idea is why we’ve learned ZPL
How does it work?

- First, the language designers, knowing the CTA, formulate operations compatible with it.
- The compiler “targets” the CTA.
- The performance of the language features is expressed in terms of CTA concepts and “given” to the programmers.
- In ZPL’s case the performance is “syntactically visible” and called the WYSIWYG performance model.
To state how ZPL performs operations, each operator’s work and communication needs are given ...

- Performance is given in terms of the CTA (and RAM)
- Performance is relative, e.g. $x$ is more expensive in communication than $y$ -- absolute not possible

- To start allocate work (owner computes) and data:

\[ P=4 \text{ allocations} \]

for 2D arrays: columns, rows, blocks
Describe the costs for all language constructs
- Declarations, control flow have negligible cost
- Scalar computations are redundant, also “free”
- Array operations costs depend on operators:

- Rules...
  - $A + B$ -- Element-wise array operations
    - No communication
    - Per processor work is comparable to C
    - Work fully parallelizable, i.e. time = work/P
A@^east -- @ references including @^  
Arrays allocated with “fluff” for every direction used

- Nearest neighbor point-to-point communication of edge elements, i.e. small communication, little congestion
- Edge communication benefits from surface-to-volume advantage: an $n$ increase in elements, adds $\sqrt{n}$ comm load
- Local data motion, possibly
<<|>>

\[\langle\langle A \rangle\rangle\] -- Reduce
- Accumulate local elements
- \(O(\log P)\) tree accumulation, or better
- Broadcast, which is worst case \(O(\log P)\), but usu. less

\[\langle| A \rangle\] -- Scan
- Accumulate local elements
- Ladner/Fischer \(O(\log P)\) tree parallel prefix logic
- Update of local elements

\[\langle\langle [1..n, k] A \rangle\rangle\] -- Flood
- Multicast array segments, \(O(\log P)\) w.c.
- Represent data “non-redundantly”
A#[I1, I2] -- Remap, both gather and scatter

- (Potential) all-to-all processors communication to distribute routing information implied by I1, I2
- (Potential) all-to-all processors communication to route the elements of A
- Heavily optimized, esp. to save first all-to-all

- Full information online in Chapter 8 of ZPL Programmer’s Guide or in dissertations
- “What you see is what you get” performance model
  ... large performance features visible

ZPL is only parallel language with performance model
More on Array Allocation

ZPL allocates regions (and therefore arrays) to processors so many contiguous elements are assigned to each to exploit locality

- **Array Allocation Rules**
  - Union the regions together to compute the *bounding region*
  - Get processor number and arrangement from the command line
  - Allocate the bounding region to the processors

Let’s walk-through the process
Create the "footprint" of the regions by aligning indices

Technical point: Only interacting regions are "unioned," e.g. if region R is used to declare an array which is manipulated in the scope of region S, R and S are said to interact.

The bounding region is allocated to processors.
The number and arrangement of processors is given by the programmer on the command line [or programmed; more later]

- For the purpose of [understanding] allocation, processors are viewed as being arranged in grids ... this is simply an abstraction:

The CTA does not favor any arrangement, so use a generic one
Allocate Bounding Region to Grid

The bounding region is allocated to processor grid in the “most balanced” way possible

- Regions inherit their position from their position in the bounding region
- Array elements inherit their positions from their index’s position in the region, and hence their allocation
More Typical Allocations

- 1D is segmented;
- 2D is panels, strips or blocks;
- 3D ...

ZPL uses Ceiling/Floor and includes fluff
Such allocations are mostly standard, but one fact makes ZPL performance clear:

ZPL has the property that for any arrays $A$, $B$ of the same rank and having an element $[i, \ldots, k]$, that element of each will be stored on the same processor.

Corollary: Element-wise operations do not require any communication: $[R] \quad \ldots \quad A + B \quad \ldots$
program Life;
config var n : integer = 512;
region R = [1..n, 1..n];
    BigR = [0..n+1,0..n+1];
direction N = [-1, 0]; NE = [-1, 1];
    E = [ 0, 1]; SE = [ 1, 1];
    S = [ 1, 0]; SW = [ 1,-1];
    W = [ 0,-1]; NW = [-1,-1];
var NN : [R] ubyte; TW : [BigR] boolean;
procedure Life();
    [R] begin
        TW := (Index1 * Index2) % 2; -- Make data
        repeat
            NN := (TW@N + TW@NE + TW@E + TW@SE
                   + TW@S + TW@SW + TW@W + TW@NW);
            TW := (NN=2 & TW) | NN=3;
            until !|<<TW;
        end;
end;
Blue: Effectively no time ... each processor does set-up and scalar computation locally

Pink: Element-wise computation perfectly parallel ... Index \( i \) constants are generated

How is TW allocated on 4 procs? Three basic choices...

Delay is c\( \lambda \)
Purple: Element-wise computation with @ operations ... expect λ delay for @ (all at once if synch’ed) and then full parallel speed-up for local operations

Red: Reduce uses Ladner/Fischer parallel prefix, with local combining and log(P) tree to communicate ... potentially the most complex operation in Life

Knowing the relative costs of the program allows us to optimize it for some purpose ... count generations
Configure and add a counter to previous program.
How Many Generations?

Testing on each generation my be excessive -- analyze

program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];
direction NW = [-1,-1]; N = [-1, 0]; NE = [-1, 1];
            W = [ 0,-1];       E = [ 0, 1];
            SW = [ 1,-1];      S = [ 1, 0]; SE = [ 1, 1];
var NN:[R] ubyte; TW:[R] boolean; count:integer = 0;
procedure Life();
    [R] begin read(TW); -- Input
    repeat
        count += 1;
        NN := (TW@^N + TW@^NE + TW@^E + TW@^SE
              + TW@^S + TW@^SW + TW@^W + TW@^NW);
        TW := (NN=2 & TW) | NN=3;
        until !|<<TW;
    writeln(count, " generations");
end;
config var n : integer = 512; epoch : integer = 50;
...
var NN:[R] ubyte; TW,TWo:[R] boolean; count:integer = 0;
procedure Live(integer:gens);
  begin var i : integer;
    for i := 1 to gens do
      NN := (TW@^N + TW@^NE + TW@^E + TW@^SE 
        + TW@^S + TW@^SW + TW@^W + TW@^NW);
      TW := (NN=2 & TW) | NN=3;
  end;
procedure Life();
  [R] begin read(TW);
    while |<<TW do
      TWo:=TW; Live(epoch); count += epoch;
    end;
    count -= epoch; TW := TWo; -- Roll back
    repeat
      Live(1); count += 1;
    until !|<<TW;
    writeln(count, " generations");
  end;
WYSIWYG, a key tool for parallel algorithm design ... work through the logic of balancing costs

- There are dozens (hundreds?) of matrix product algorithms ... which do you want?

MM is a common building block, so someone else should have done this (vdG&W did!) but we use it as an example of process

- Two popular choices are
  - Cannon’s algorithm
  - SUMMA
- Which is better as a ZPL program, i.e. better for the CTA model
Cannon’s Algorithm: MM in Motion

Compute: $C = AB$ as follows ...

$C$ is initialized to 0.0

Arrays $A$, $B$ are *skewed*

$A$, $B$ move “across” $C$ one step at a time

Elements arriving at a place are multiplied, added in
Motion of Cannon’s Algorithm

Step 1

\[
\begin{align*}
\begin{array}{ccc}
    c_{11} & c_{12} & c_{13} \\
    c_{21} & c_{22} & c_{23} \\
    c_{31} & c_{32} & c_{33} \\
    c_{41} & c_{42} & c_{43} \\
\end{array}
\end{align*}
\begin{align*}
    & a_{11} \ a_{12} \ a_{13} \ a_{14} \\
    & a_{21} \ a_{22} \ a_{23} \ a_{24} \\
    & a_{31} \ a_{32} \ a_{33} \ a_{34} \\
    & a_{42} \ a_{43} \ a_{44} \\
\end{align*}
\]

\[
\begin{aligned}
    c_{43} &= c_{43} + a_{41} b_{13} \\
    c_{42} &= c_{42} + a_{41} b_{12} \\
\end{aligned}
\]

Second steps ...

\[
\begin{align*}
    c_{43} &= c_{43} + a_{42} b_{23} \\
    c_{33} &= c_{33} + a_{31} b_{13} \\
\end{align*}
\]

60
Pack skewed arrays into dense arrays by rotation; process all $n^2$ elements at once.
Four Steps of Skewing A

for i := 2 to m do
[i..m, 1..n] A := A^[right]; -- Shift last m-i rows left
end;

... And Skew B vertically

a11 a12 a13 a14
a21 a22 a23 a24
a31 a32 a33 a34
a41 a42 a43 a44

Initial
a11 a12 a13 a14
a22 a23 a24 a21
a33 a34 a31 a32
a43 a44 a41 a42

i = 3 step

a11 a12 a13 a14
a22 a23 a24 a21
a33 a34 a31 a32
a44 a41 a42 a43

i = 4 step

a11 a12 a13 a14
a22 a23 a24 a21
a33 a34 a31 a32
a44 a41 a42 a43

i = 2 step
For completeness, when $A$ is $m \times n$, $B$ is $n \times p$, and the declarations are ...

region
Lop = [1..m, 1..n];
Rop = [1..n, 1..p];
Res = [1..m, 1..p];
direction right = [0, 1];
below = [1, 0];
var
A : [Lop] double;
B : [Rop] double;
C : [Res] double;
Cannon’s Algorithm

Skew A, Skew B, \{Multiply, Accum, Rotate\}

\[
\text{for } i := 2 \text{ to } m \text{ do } -- \text{ Skew A}
\]
\[
[i..m, 1..n] A := A[^\text{right}];
\]
\[
\text{end;}
\]
\[
\text{for } i := 2 \text{ to } p \text{ do } -- \text{ Skew B}
\]
\[
[1..n, i..p] B := B[^\text{below}];
\]
\[
\text{end;}
\]
\[
[\text{Res}] C := 0.0; -- \text{ Initialize C}
\]
\[
\text{for } i := 1 \text{ to } n \text{ do } -- \text{ For common dim}
\]
\[
[\text{Res}] C := C + A*B; -- \text{ For product}
\]
\[
[\text{Lop}] A := A[^\text{right}]; -- \text{ Rotate A}
\]
\[
[\text{Rop}] B := B[^\text{below}]; -- \text{ Rotate B}
\]
\[
\text{end;}
\]
SUMMA Algorithm in ZPL

```zpl
var Col : [1..m,*] double; -- Col flood array
Row : [*,1..p] double; -- Row flood array
A : [1..m,1..n] double;
B : [1..n,1..p] double;
C : [1..m,1..p] double;

... 
[1..m,1..p] C := 0.0; -- Initialize C
for k := 1 to n do
  [1..m,*] Col := >>[ ,k] A; -- Flood kth col of A
  [*,1..p] Row := >>[k, ] B; -- Flood kth row of B
  [1..m,1..p] C += Col*Row; -- Combine elements
end;
```
Comparing Cannon’s & SUMMA MM

- Analyze the choices with WYSIWYG ...
  - SUMMA has shortest code [so what?]
  - Cannon’s uses only local communication
- The two algorithms abstractly:

**Cannon’s**
- Declare
- Skew A
- Skew B
- Initialize
- loop til n
- C+=A*B
- Rotate A,B

**SUMMA**
- Declare
- Initialize
- loop til n
- Flood A
- Flood B
- C+=A*B
Comparing Cannon’s and SUMMA MM

- Step one is to cancel out the equivalent parts of the two solutions ... they’ll work the same
- For MM the comparison reduces to whether the initial skews and the iterated rotates are more/less expensive than iterated floods
Cannon’s Algorithm

Skew A, Skew B, \{Multiply, Accum, Rotate\}

\[
\text{for } i := 2 \text{ to } m \text{ do -- Skew A} \\
\quad [i..m, 1..n] \ A := A@^\text{right}; \\
\quad \text{end; }
\]

\[
\text{for } i := 2 \text{ to } p \text{ do -- Skew B} \\
\quad [1..n, i..p] \ B := B@^\text{below}; \\
\quad \text{end; }
\]

\[
\text{[Res]} \ C := 0.0; \quad \text{-- Initialize C} \\
\quad \text{for } i := 1 \text{ to } n \text{ do -- For common dim} \\
\quad \quad \text{[Res]} \ C := C + A*B; \quad \text{-- For product} \\
\]

\[
\text{[Lop]} \ A := A@^\text{right}; \quad \text{-- Rotate A} \\
\text{[Rop]} \ B := B@^\text{below}; \quad \text{-- Rotate B} \\
\quad \text{end; }
\]

Comms have \(\lambda\) latency, but much data motion
The flood is (likely) more expensive than $\lambda$ time, but less that $\lambda(\log P)$ ... probably much less, and there are fewer of them

```
[1..m,1..p] C := 0.0; -- Initialize C
for k := 1 to n do
  [1..m,*] Col := >>[ ,k] A; -- Flood kth col of A
  [*,1..p] Row := >>[k, ] B; -- Flood kth row of B
  [1..m,1..p] C += Col*Row; -- Combine elements
end;
```

SUMMA does not require as much comm or data motion as Cannon’s, nor does it “touch” the array as much
We assert that SUMMA is the better algorithm
- Though it does “potentially more expensive” communication, it does less of it
- It’s “nonredundant” flood arrays cache well
- There is less data motion

Analytically ...

<table>
<thead>
<tr>
<th>algorithm</th>
<th>number of communications</th>
<th>communication complexity</th>
<th>communication volume</th>
<th>flops</th>
<th>elements referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannon</td>
<td>$4n$</td>
<td>$1$</td>
<td>$n$</td>
<td>$2n^3 - n^2$</td>
<td>$n \cdot (2n^2/2 + 3n^2)$</td>
</tr>
<tr>
<td>SUMMA</td>
<td>$2n$</td>
<td>$\log p$</td>
<td>$n$</td>
<td>$2n^3$</td>
<td>$n \cdot (n^2 + 2n)$</td>
</tr>
</tbody>
</table>

Y-axis is time …

- **SUMMA**
- **Cannon**
Optimizations Can Help

- WYSIWYG is the worst case ... optimizations are possible ...
- **Sequential Optimizations** e.g. stencil opts

Sum of orange items performed once

- **Parallel Optimizations** e.g. communication motion -- prefetching to overlap communication with computation

7 additions are used for each element, but fewer adds are sufficient
Guarantees

ZPL uses a different approach to performance than other parallel languages

- *Historically, performance came from compiler optimizations that might/might not fire ...*
- WYSIWYG guarantees (it’s a contract) that ZPL programs will work a certain way ...
  - It may be better ... WYSIWYG is a worst case that often doesn’t materialize
  - Aggressive optimizations help a lot

If there are any surprises, they’ll be pleasant
Data and processing allocations are given
- All constructs of the language are explained in terms of the allocations and the CTA
- Result: relative, worst-case statement of how the computation runs anywhere ... rely on it
- Optimizations can improve on the times, but if they don’t fire, nothing is lost

The best use of the WYSIWYG model is to make comparative programming decisions
The reason we learned ZPL was because it illustrates how a high level parallel language can give access to the CTA machine model, allowing programmers to write intelligent parallel programs easily and portably.

- You want your programming language to have that property, too!
- If it doesn’t, dump it and use a library that lets you apply the CTA model yourself.
ZPL Classic (the portion we learned) is a global view language, meaning it’s

- *P-independent*, all executions of the program produce the same result regardless of the number or arrangement of the processors
  
- Functional languages tend to be P-independent
  
- P-independent is a very desirable property from a programmers view

- Another is NESL
NESL

- NESL was developed by Guy Belloch at CMU
- Key structure is a sequence
  - \([2, 14, -5, 0, 7]\)
  - "sequences can be composed of characters"
  - ["sequence" "elements" "can be sequences"]
    provided all are composed of the same atomic type
- Basic operation is apply to each, written with set notation
  \[\{a+1: a \in [2, 13, 0, 4, 8]\} \text{ producing } [3, 14, 1, 5, 9]\]
  \[\{a+b: a \in [1, 2, 3]; b \in [8, 7, 6]\} \text{ producing } [9, 9, 9]\]
More on NESL

- Compare NESL dot product with UPC

```plaintext
function dotprod(a, b) = sum({x*y: x in a; y in b});
dotprod([2, 3, 1], [6, 1, 4]);
```
producing [19]

- “Nested” in NESL refers to nested parallelism:
  - Applying parallelism and within each parallel operation, applying more parallelism
  - In NESL, *apply to each* ops in *apply to each*
  - Consider NESL’s matrix multiplication algorithm
The function is defined

```javascript
function matrix_multiply(A,B)=
{{sum({x*y : x in rowA; y in columnB})
 : columnB in transpose(B)}
 : rowA in A}
```

Three *apply to each* braces
- Outer brace applied to rowA, in ||
- Next brace applied to columnB, transposed, in ||
- Inner brace applied to each of \(n^2\) row/col pairs
NESL researchers identify two types of complexity in a program:

- **Work**, which is the number of basic operations
  - MM has $O(n^3)$ work; dotproduct has $O(n)$ work
- **Depth**, which is the longest chain of dependences; e.g. sum has $O(\log_2 n)$ depth
  - Both MM and dotproduct have $O(\log_2 n)$ depth

Like the PRAM, these metrics do not yield a performance model as they are not conditioned on $P$, $\lambda$, locality, etc.
Parallel programming will be convenient and non-disruptive when languages provide the kinds of abstractions programmers need.

ZPL abstracts above the HW, but not so far that we loose track of the underlying (logical) machine.

- ZPL achieves performance-portability
- ZPL “works” because it has a built-in performance model: WYSIWYG
- You use a performance model – it might as well be one that the compiler-writers target
Homework 7

- Write a ZPL program to solve the Red/Blue simulation using a new termination criterion, and analyze its performance w.r.t. WYSIWYG model
  - Terminate if any row or column outside the range [too few, too many] (N.B. *This is different from the earlier assignment.*)
  - Classify the statements in terms of their approximate cost using the WYSIWYG model
  - Submit a document with program & analysis